University of Zurich[UZH]

Zurich Open Repository and Archive
University of Zurich
University Library
Strickhofstrasse 39
CH-8057 Zurich
www.zora.uzh.ch

Year: 2018

Cutter – a Universal Multilingual Tokenizer

Graën, Johannes ; Bertamini, Mara ; Volk, Martin

Abstract: Tokenization is the process of splitting running texts into minimal meaningful units. In writing systems where a space character is used for word separation, this blank character typically acts as token boundary. A simple tokenizer that only splits texts at space characters already achieves a notable accuracy, although it misses unmarked token boundaries and erroneously splits tokens that contain space characters. Different languages use the same characters for different purposes. Tokenization is thus a language-specific task (with code-switching being a particular challenge). Extralinguistic tokens, however, are similar in many languages. These tokens include numbers, XML elements, email addresses and identifiers of concepts that are idiosyncratic to particular text variants (e.g., patent numbers). We present a framework for tokenization that makes use of language-specific and language-independent token identification rules. These rules are stacked and applied recursively, yielding a complete trace of the tokenization process in form of a tree structure. Rules are easily adaptable to different languages and text types. Unit tests reliably detect if new token identification rules conflict with existing ones and thus assure consistent tokenization when extending the rule sets.

Posted at the Zurich Open Repository and Archive, University of Zurich
ZORA URL: https://doi.org/10.5167/uzh-157243
Conference or Workshop Item
Published Version

The following work is licensed under a Creative Commons: Public Domain Dedication: CC0 1.0 Universal (CC0 1.0) License.

Originally published at:
Graën, Johannes; Bertamini, Mara; Volk, Martin (2018). Cutter – a Universal Multilingual Tokenizer. In: Swiss Text Analytics Conference, Winterthur, 12 June 2018 - 13 June 2018. CEUR-WS, 75-81.

Cutter – a Universal Multilingual Tokenizer

Johannes Graën, Mara Bertamini, Martin Volk
Institute of Computational Linguistics
University of Zurich
graen@cl.uzh.ch, bertaminimara@gmail.com, volk@cl.uzh.ch

Abstract

Tokenization is the process of splitting running texts into minimal meaningful units. In writing systems where a space character is used for word separation, this blank character typically acts as token boundary. A simple tokenizer that only splits texts at space characters already achieves a notable accuracy, although it misses unmarked token boundaries and erroneously splits tokens that contain space characters.

Different languages use the same characters for different purposes. Tokenization is thus a language-specific task (with code-switching being a particular challenge). Extra-linguistic tokens, however, are similar in many languages. These tokens include numbers, XML elements, email addresses and identifiers of concepts that are idiosyncratic to particular text variants (e.g., patent numbers).

We present a framework for tokenization that makes use of language-specific and language-independent token identification rules. These rules are stacked and applied recursively, yielding a complete trace of the tokenization process in form of a tree structure. Rules are easily adaptable to different languages and test types. Unit tests reliably detect if new token identification rules conflict with existing ones and thus assure consistent tokenization when extending the rule sets.

In: Mark Cieliebak, Don Tuggener and Fernando Benites (eds.): Proceedings of the 3rd Swiss Text Analytics Conference (SwissText 2018), Winterthur, Switzerland, June 2018

CEUR-WS.org/Vol-2226/paper9.pdf

1 Introduction

Common wisdom has it that tokenization is a solved problem. Yet, in practice, we often find ourselves in bothersome trials of adapting tokenizers or their output. This may be due to the fact that "down-stream" processing tools require a different tokenization. Or it may be because of special tokenization needs for particular domains, genres or historical text variants.

As an example of different tokenization needs, consider splits of English negations in contracted forms like didn't and won't. The Penn Treebank guidelines suggest to tokenize those as did + n't and wo + n't. Such splits are practical for information extraction or sentiment analysis. But, of course, these splits make searching a corpus (e.g. for linguistic investigations) for negated forms unintuitive. Searches must then be supported by a specific module that undoes the splits.

Another example is the English phrase a 12-ft boat. How shall we handle the hyphenated length expression? Is this one or two or even three tokens? We follow the rule that measurement units are split from numerical values. This rule is meant for altitude or speed and says that the number is split from the unit (e.g. 2850m → 2850, m; 155km/h → 155, km/h). Following this rule, we decided to also split the hyphenated length expression into two tokens resulting in: a, 12, -ft, boat. Once identified as such, we can, of course, keep numerical values and measurement units as single tokens, if required by the following processing step.

We work on the annotation of large multilingual corpora, some of them diachronic for the last 150 years. In our work such tokenization issues abound. We have therefore developed tokenization guidelines which started out as check-lists for the various language versions of our corpora. We then realized that only a custom-built tokenizer with systematic tests in-

cluded will serve our purposes of high-quality tokenization.

Our tokenization approach does not include normalization, which we see as a separate step involving coding issues (like turning ligatures into letter sequences, or certain spaces into non-breakable spaces) or other simplifications (like turning American into British English spelling, or Swiss German into Standard German spelling).

In this paper, we first describe existing tokenization approaches and show that there is a need for tokenizers that can be adapted to particular language and text variants (Section 2). We then show why tokenization is a challenging task by giving examples of ambiguous cases. We argue that a tokenizer needs to possess linguistic information and to consider long-distance relations to be able to decide those cases (Section 3).

Having outlined the problem, we describe our tokenization approach (Section 4), and how we employ unit testing to warrant high-quality tokenization while allowing for the adaptation of the tokenizer (Section 5). Finally, we show that there are cases where tokenization decisions require commonsense knowledge, and which our tokenizer is not capable to handle (Section 7). Future development (Section 8) will need to involve syntactic parsing to solve those hard cases.

2 Related Work

The Stanford Tokenizer (Manning et al., 2018) is probably the most widely used tokenizer for English. It is built on the basis of the tokenization rules in the Penn Treebank.[1] Following the Penn tokenizations gets us a long way for English, but is not explicit enough to address issues such as in the hyphenated length expressions above, a 12-ft boat.

Its strength is its speed and the numerous options concerning the treatment of special symbols (parentheses, ampersand, currency symbols and fractions). In contrast, our tokenizer is highly modular and adaptable to categories of texts that we did not consider when compiling our guidelines. It also allows for a combination of rule sets from different languages to process texts with quotations or code switching, for instance.

He and Kayaalp (2006) compare various tokenizers for the biomedical domain. Their results point to the need for standard tokenizers in order to ensure the interoperability of processing tools. Cruz Díaz and Maña López (2015) follow up with an analysis of more recent tokenizers also for the biomedical domain. They observe disagreement to a large extend between the tokenization decisions of those tools for the test cases they had identified preliminarily. That observation is in agreement with Habert et al. (1998), when they concluded more than 15 years earlier "At the moment, tokenizers represent black boxes, the behavior and rationale of which are not made clear."

Apart from rule-based tokenization, there are machine learning approaches to tokenization as well. For those approaches, a certain amount of training material (i.e., both original and tokenized versions of the same texts) is required. Jurish and Würzner (2013) argue that sufficient training material could be extracted from "treebanks or multi-lingual corpora".

3 Tokenization Challenges

Although the only decision to be taken by the tokenizer is whether or not to place a token boundary between each two adjacent characters, this task is not as trivial as it seems at first glance. If two adjacent characters are both letters, they typically belong to the same token. English negations in contracted forms (like didn't) as described above are one exception.

A non-letter character (e.g., a punctuation mark) followed by a letter frequently marks the boundary of two tokens, while the opposite case (a letter character followed by a non-letter character) does not show a general preference; the right decision in these cases often requires to resort to linguistic knowledge. We can, for instance, not decide if baby's is one token or two without knowing whether a text is written in English ('s is a possessive marker of baby) or Dutch (baby's means *babies*).

Apart from knowing a text's language, which includes word formation and grammar knowledge, sometimes long-distance relations between tokens that belong together, such as brackets or quotation marks, have to be determined in order to take the right decision. An apostrophe following a German word that phonetically ends in /s/ can be both a possessive marker or the end of a single-quoted expression. If we find another apostrophe in the same sentence preceding the ambiguous one such that it is followed immediately by a letter-character and preceded by a space

[1] ftp://ftp.cis.upenn.edu/pub/treebank/public_html/tokenization.html

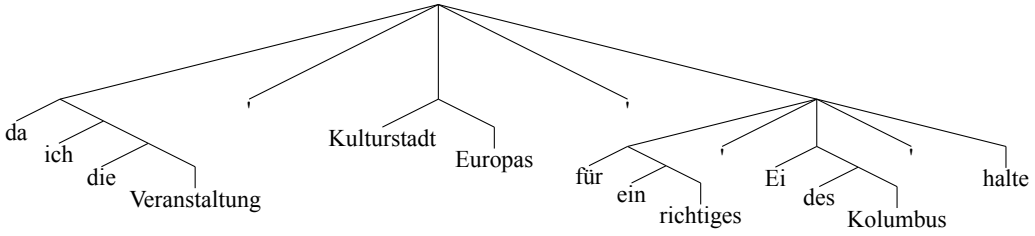

Figure 1: German sub-clause “da ich die Veranstaltung 'Kulturstadt Europas' für ein richtiges 'Ei des Kolumbus' halte” with typewriter apostrophes as tokenization tree. Every node represents a single decision of the tokenizer. Example taken from (Graën, 2018, p. 31).

character, we have evidence for the quoted expression and consequently mark both apostrophes as single tokens (see Figure 1).[2]

In many languages, both sentences and abbreviations typically end with a period. While the sentence-final period is a single token, abbreviations comprise the period. To distinguish both cases, we either need to know all abbreviations of the language in questions. Or we need a reliable way of determining sentence boundaries.

4 The Cutter Implementation

Simple tokenizers process text as a stream of characters from left to right and take locally justified decisions of whether to place a token boundary between two adjacent characters. This approach is limited as it is not capable of taking long-distance relations into account.

Our approach is to successively identify tokens following an ordered list of patterns defined by advanced regular expressions.[3] Once identified, we ‘cut out’ the token (hence the name Cutter) and proceed by applying the same patterns to the remaining parts, until only empty character sequences remain. This procedure generates a tree structure like the ones in Figure 1 and Figure 2.

The order of patterns that describe tokens and their respective context is chosen such that the more detailed or exceptional tokens are identified first, followed by more common and standard tokens. That way, sequences of characters that would otherwise be split into several tokens can be protected by an earlier match, which prevents that sequence from further processing.

Tokens that contain spaces, for instance, need to be matched by a pattern that prevents them from being split by the general rule which mandates that spaces (and other white space characters) are token separators. For ease of reading, numbers are often separated into groups of three digits. The international standard for “quantities and units” stipulates the use of a small space as separator (ISO 80000-1, 2009, Section 7.3.1), which is often realized as a standard space in electronic texts. We identify numbers formatted in this way (e.g., 50 000) as single tokens.

Another example for tokens that need to be protected are French words originally composed of more than one lexical unit that nowadays form a single lexical unit and should thus be recognized as a single token. In the example shown in Figure 2, aujourd’hui *‘today’* is identified as token in the first step, leaving On nous dit qu’ and c’est le cas, encore faudra-t-il l’évaluer. as remainders, which are subsequently further tokenized. To be able to distinguish lexicalized forms (e.g., d’accord → d’accord) from regular elision of vowels (e.g., d’accorder → d’, accorder), we need to incorporate all lexicalized forms (e.g., entr’ouvèrt, c’est-à-dire, presqu’île) into the patterns that constitute our (tokenization) language model.

In addition to the linguistic information encoded in patterns, our tokenizers uses two word lists per language. The first one has abbreviations in order to mark their occurrences in the text as single tokens. The second one consists of sentence-initial words, that is, words that do not start with a capital letter except in sentence-initial position, such as preposition or deter-

[2] This is only necessary if the single typewriter apostrophe is used instead of the proper left and right single quotation marks.

[3] We use the so-called Perl Compatible Regular Expressions (PCRE) by (Hazel, 1997), including adjuvant features, such as Unicode character properties (The Unicode Consortium, 2017), named capturing subpatterns and subpattern assertions.

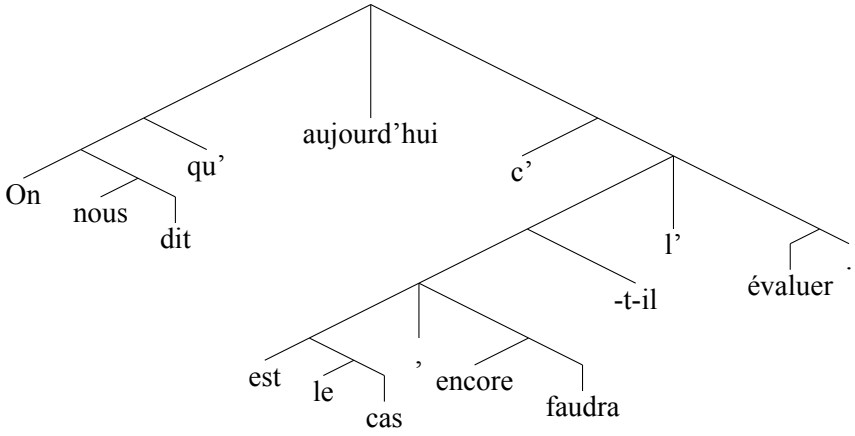

Figure 2: Tokenization tree of the French sentence “On nous dit qu’aujourd’hui c’est le cas, encore faudra-t-il l’évaluer.”. Example taken from (Graën, 2018, p. 31).

miners. When we locate a word from this list in the text, we mark it as potential sentence starter, which in particular contexts leads to a special empty token that marks a sentence boundary. Sentences can be split at those markers, if no prior sentence segmentation has been performed.[4]

Pattern identification rules are composed of a list of named capturing subpatterns (see Hazel, 1997) that extend to the whole text provided (i.e., they are anchored both at the beginning and at the end of the text). We distinguish between the actual token or tokens that are identified by a rule (e.g., , in Figure 2) and the remainders that await further processing (est le cas and encore faudra in Figure 2). A typical rule consists of a left part, the actual token and a right part (see root node in Figure 2), though it can also identify more tokens (see root node in Figure 1).

Some examples: Date expressions, for instance, typically consist of a number of tokens (e.g., 12. , und , 13. , Juni , 2018); pronouns in some Romance languages can be concatenated (e.g., No vull posar-n’hi. → No vull posar , -n’ , hi , . ; Sim, dir-lhes-ia isso. → Sim, dir , -lhes , -ia , isso.). Each identified pattern is assigned a tag, which marks the corresponding language (if the rule is language-dependent), the rule name, a running letter (if there is more than one rule for the same target) and a running number (if a rule identifies more than one token).

[4] If a sufficiently large corpus exists for the language and text variant in question, methods to learn sentence boundaries from the corpus such as (Kiss and Strunk, 2006) may perform better.

We envisage that our Cutter applies the language-independent rules together with the rules for a particular language to a text whose language is known and uniform. Beyond that, rule sets of different languages can be combined in case of code switching. Various rule sets for the same language (e.g., for different text variants) can also be combined individually.

5 Unit Testing

The architecture of our tokenizer has a modular design to facilitate its adaptation to different user needs. Towards this goal, we need to check that new rules do not interfere with existing ones. Following the method of unit testing, widely-used in software development, we collect text snippets for each nontrivial tokenization problem. We then provide information on correct tokenization of those snippets according to our guidelines. To make a unit test pass, our tokenizer needs to perform tokenization exactly like indicated.

If a unit test fails, the error can be either in the test or in the rules. A test error can be based on contradictory or unachievable tokenization guidelines (i.e., requiring commonsense knowledge) or an incomplete manual tokenization (e.g., the annotator missed a comma). A rule error typically results from a rule being too restrictive or another rule being too general and thus erroneously matching the unit test in question.

In both cases, iterative improvement of tests or rules (or both) finally leads to a configuration where all tests pass, which is the objective of unit testing and, in our case, guarantees that a deterioration of tokenization quality by reason of language model changes are de-

tected immediately and can be traced back to a particular change.

6 Evaluation

Aside from the implicit evaluation of our unit tests, all of which we require to pass, we tested the performance of our tokenizer on gold-tokenized texts. Such gold-standard texts together with their original (untokenized texts) are not easy to obtain. Corpora are typically available as raw texts, while treebanks typically feature the manually determined tokens, but not the original, untokenized material. Jurish and Würzner (2013) approach this problem with de-tokenization rules and a manual correction in particular cases.

We built a test corpus based on the SMULTRON treebanks (Gustafson-Capková et al., 2007; Volk et al., 2010), for which we have the original texts. Using those treebanks for comparison with the output of our tokenizer is problematic, however, since the tokenization guidelines that our tokenizer implements originate, among others, from the experiences in the creation of these very treebanks. Notwithstanding the expected bias, we select those sentences from the treebanks which we can identify in the original texts (1528 unique sentences in total) by simply ignoring any whitespace characters.[5] This selection comprises 1173 sample sentences in four languages: English (67), German (388), Spanish (184) and Swedish (534).

An initial tokenization with Cutter yields 59 sentences with errors (0.5 %).[6] In more than half of these cases, the tokenization in the treebank deviates from the pattern stipulated by the tokenization guidelines. Another frequent issue is that the textual source does not correspond to the text in the original document, which comprises missing or superfluous whitespaces and the representation of images as characters (see Figure 3).

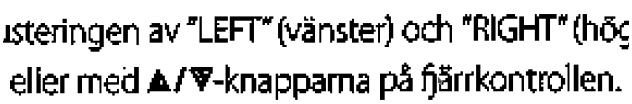

Figure 3: The ‘up’ and ‘down’ key symbols in this detail from a DVD player manual are represented as circumflex diacritic and letter ‘v’, respectively: eller med ^/v-knapparna på fjärkontrollen.

[5] From our point of view, a tokenizer is only allowed to remove input characters, not to alter them.

[6] Only the first error found in a sentence is counted.

We correct apparent anomalies in the input sentences and remove sentences that cannot regularly be represented as text, which are all key symbols, such as the one shown in Figure 3. That way, we obtain a small tokenization gold standard. It comprises 1165 sentences in the aforementioned four languages.

When we tokenize the tests obtained from the gold standard sample sentences with our tokenizer, we still see an error rate of 1 %. By adjusting the existing rules to include borderline cases (e.g., including the ± sign into the definition of numbers), we could make all tests pass. The error rate of two other popular tokenizers, the ones in the NTLK[7] and the Spacy[8] NLP toolkits, is at approximately 12 %.[9] The comparatively high error rate is due to both real tokenization errors, such as splitting URLs, XML tags and ordinal numbers in German,[10] and, of course, debatable tokenization rules. Should the German adjective 100%ige be left as one token (Spacy), or be split into two tokens (100 , %ige ; Cutter) or three tokens (100 , % , ige ; NLTK)?

7 Features and Limitations

Rule-based approaches in natural language processing have widely been replaced by machine learning approaches, since the latter are capable of handling unanticipated situations by abstraction from observed patterns. For tokenization of standard texts in well-resourced languages, machine learning approaches such as (Jurish and Würzner, 2013) might have enough data from which to learn those patterns. For particular text categories and low-resourced languages, however, providing the algorithm with sufficient training data will require a substantial effort.

In our work with corpora in several languages, the best approach turned out to be an iterative one. For a new language, we start with an empty rule set (in addition to the language-independent rules) and apply it to the untokenized texts. We subsequently generate unit tests from the errors that surfaced in manual inspection, which we then address by defining corresponding patterns (also consulting grammar books and treebanks, if available). Few iterations of this proce-

[7] https://www.nltk.org/

[8] https://spacy.io/

[9] The Spacy tokenizer has only been evaluated on English, German and Spanish sentences as it has no model for Swedish.

[10] The Spacy tokenizer also consequently splits compound adjectives and nouns in English (e.g., low-cost, medium-voltage, break-even), while the NLTK tokenizer alters all quotation marks.

dure lead to a collection of rules and tests, and lower the tokenization error rate considerably.

For languages that have treebanks or sentence-segmented corpora, we can automatically extract sentence-initial words and add them to our list, if they do not appear with a capital letter in other positions; we might also want to filter for closed word classes (i.e., preposition, pronouns, etc.) here. If no resource for gathering abbreviations is available, we need to search for abbreviations in the given texts.

In contrast to machine learning approaches, erroneous tokenization decisions in our system can always be traced back to a particular pattern, which facilitates a quick remedy. The language model, however, is inevitably incomplete and requires testing and adaptation ahead of its application to new text variants.

As mentioned above, some tokenization decisions require a deeper understanding than what a sequence of characters can provide. This is, for instance, the case when abbreviations (without the period) coincide with another word. We are only aware of German examples such as Abt. for *Abteilung 'department'* vs. Abt *'abbot'* or Art. for *Artikel 'article'* vs. Art *kind*. Even if we address this problem by excluding those words from the abbreviation list and match them with a dedicated rule that expects a succeeding number (e.g., in Abt. 3 *'in department 3'*; nach Art. 25 *'pursuant to article 25'*), we can still come up with cases that cannot be solved without dictionary lookups or parsing. Compare, for instance:

1. *Wir trafen den Abt. Bergbahnen sind seine Leidenschaft.*
   'We met the abbot. Mountain railways are his passion.'

2. *Wir sahen den Sprecher der Abt. Bergbahnen und Wanderwege.*
   'We saw the spokesman of the dept. of mountain railways and hiking trails.'

Splitting undirected quotation marks (") results in an information loss.[11] After tokenization, it can no longer be inferred whether such a quotation mark signals the beginning or the end of a quotation. A more careful tokenizer needs to preserve the information whether the quotation mark was split from the previous or from the following word. Our tokenizer provides the option to alternatively return or suppress white space tokens.[12]

8 Conclusions and Future Development

To overcome ambiguous cases, we propose to extend the shallow processing of the tokenizer by a syntactic parser, to select the more likely tokenization. To this end, tokenization has to be performed several times with alternating rules. Parsing likelihood as decision maker is only required if different results are obtained. For low-resourced languages where no parser exists, a heuristic based on the identification of finite verb forms could suffice.

Rules are currently organized in sets, one for each language and one for language-independent rules. Each set comprises different stages, which are used to interconnect different sets. Corresponding quotation marks, for instance, need to be identified before any token in between them splits the sentence into smaller parts. Language-specific date expressions (e.g., with ordinal numbers expressed as digits plus a period) need to be processed before the language-independent identification of numbers takes place.

We think that instead of a limited list of stages, a more dynamic data structure would be beneficial. We already know which rules interfere (e.g., numbers with spaces vs. spaces as separators), but this is not explicitly reflected in the data. If we were to reorganize the tokenization rule sets by means of a "before" relation between pairs of rules, we could build a rule dependency graph, which, serialized, would define the order of rules to apply. In case of code-switching sentences, the order of languages given would be decisive if no order is enforced by that graph.

We have used the tokenizer ourselves in a number of projects. It supports several European languages, including Romansh as a low-resourced language, and more languages are in preparation. The tokenizer and all our language models are freely available.[13] We also provide a web demo and a tokenization web service.

[11] The same is true for typewriter apostrophes (') as a replacement of matching single quotation marks.

[12] A sample sentence with whitespace tokens looks like this: Suot , , il , , titel , , " , vacanzas , , e , , cultura , " , , as , , prouva , , d' , eruir , , la , , funcziun , , da , , la , , lingua , , e , , cultura , , rumauntscha , , per , , il , , turissem , , i , 'l , , Grischun , .

[13] http://pub.cl.uzh.ch/purl/cutter

Acknowledgments

This research was supported by the Swiss National Science Foundation under grant 105215_146781/1 through the project "SPARCLING – Large-scale Annotation and Alignment of Parallel Corpora for the Investigation of Linguistic Variation".

References

Cruz Díaz, Noa Patricia and Manuel Jesús Maña López (Sept. 2015). "An Analysis of Biomedical Tokenization: Problems and Strategies". In: *Proceedings of the Sixth International Workshop on Health Text Mining and Information Analysis*. Lisbon, Portugal: Association for Computational Linguistics, pp. 40–49.

Graën, Johannes (2018). "Exploiting Alignment in Multiparallel Corpora for Applications in Linguistics and Language Learning". PhD thesis. University of Zurich.

Gustafson-Capková, Sofia, Yvonne Samuelsson, and Martin Volk (2007). *SMULTRON – The Stockholm MULtilingual parallel Treebank*.

Habert, Benoit, Gilles Adda, M. Adda-Decker, P. Boula de Marëuil, S. Ferrari, O. Ferret, G. Illouz, and P. Paroubek (1998). "Towards tokenization evaluation". In: *Proceedings of the 1st International Conference on Language Resources and Evaluation (LREC)*. Vol. 98, pp. 427–431.

Hazel, Philip (1997). *PCRE (Perl-compatible regular expressions)*. URL: https://www.pcre.org/.

He, Ying and Mehmet Kayaalp (2006). *A Comparison of 13 Tokenizers on MEDLINE*. Tech. rep. U.S. National Library of Medicine, Lister Hill National Center for Biomedical Communications.

ISO 80000-1 (Nov. 2009). *ISO 80000-1: Quantities and units – Part 1: General*. Ed. by ISO/TC 12 Technical Committee.

Jurish, Bryan and Kay-Michael Würzner (2013). "Word and Sentence Tokenization with Hidden Markov Models". In: *Journal for Language Technology and Computational Linguistics* 28.2, pp. 61–83.

Kiss, Tibor and Jan Strunk (2006). "Unsupervised Multilingual Sentence Boundary Detection". In: *Computational Linguistics* 32.4, pp. 485–525.

Manning, Christopher, Tim Grow, Teg Grenager, Jenny Finkel, and John Bauer (Aug. 2018). *Stanford Tokenizer*. URL: http://nlp.stanford.edu/software/tokenizer.shtml.

The Unicode Consortium (2017). *The Unicode Standard, Version 10.0*.

Volk, Martin, Anne Göhring, Torsten Marek, and Yvonne Samuelsson (2010). *SMULTRON (version 3.0) – the Stockholm MULtilingual parallel TReebank*. An English-French-German-Spanish-Swedish parallel treebank with sub-sentential alignments.